Algorithms and Software for Solving Finite Element Equations on Serial and Parallel Architectures

Alan George

Dept. of Computer Science, University of Tennessee

and

Mathematical Sciences Section, Oak Ridge National Laboratory

Abstract

Over the past 15 years numerous new techniques have been developed for solving systems of equations and eigenvalue problems arising in finite element computations. A package called SPARSPAK has been developed by the author and his co-workers which exploits these new methods. The broad objectives of this research project is to incorporate some of this software in the CSM testbed, and to extend the techniques for use on multiprocessor architectures. ~~The work is being supported by NASA grant NAG-1-803.~~

General Objectives

- Develop and test algorithms and software for solving finite element systems of equations.
- Install software in the CSM testbed, and conduct comparisons of new algorithms with those already in the testbed.
- Extend or adapt the algorithms for use on parallel architectures.

General Objectives

The general objective of the project is to perform research into sparse matrix techniques for solving systems of equations and eigenvalue problems arising in finite element computations. The research is to be done in the context of the CSM testbed employed by the Structural Mechanics Branch at the Langley Research Center. In addition to providing new and hopefully better capabilities for users of the testbed, the automatic generation of realistic structural mechanics problems will provide a good environment in which to develop new algorithms for solving sparse systems.

The proposed research is to be conducted over a period of two years. In the first year of the grant, which began Sept. 1, 1987, existing sparse matrix software developed by the principal investigator and his co-workers will be modified and installed in the CSM testbed, and performance comparisons with existing sparse matrix techniques already in the testbed will be conducted. In addition, the application of some recently developed sparse matrix techniques for handling indefinite problems will be explored in connection with solving eigenvalue problems generated by the testbed.

In the second year of the proposed research program, the focus of the investigation will be on extending the basic techniques to exploit multiprocessor architectures, again in the context of the CSM testbed. Ideas and software already developed or in development are to be adapted and incorporated into the testbed, and performance studies will be conducted.

Specific Tasks

- Modify and install some software from the sparse matrix package SPARSPAK in the CSM testbed. Status: testbed installed on SUN workstation.
- Compare performance with existing sparse matrix techniques already in the testbed.
- Application of some new techniques for solving *indefinite* problems to the solution of eigenvalue problems generated by the testbed. Status: basic algorithm and software developed.
- Extend the methods employed in SPARSPAK for use on multiprocessors.
- Compare performance with parallel methods already available in the testbed.
- Develop new algorithms and software, particularly in the area of automatic mapping of tasks to processors.

Specific Tasks

An initial objective will be to compare the performance of state-of-the-art techniques for solving sparse systems with those that are currently available in the CSM testbed. Thus, one of the early tasks will be to become familiar with the structure of the testbed, and to install some or all of the SPARSPAK package in the testbed. This will allow performance studies to be conducted. To date, the CSM testbed has been installed on a network of SUN workstations at the University of Tennessee, and the demonstration problems have been run successfully. Studies on how best to integrate SPARSPAK into the testbed are currently in progress. Installation of the testbed uncovered a few minor bugs in the install procedure, but apart from that, the process of installing the testbed in the UNIX environment is essentially automatic and very straightforward.

The testbed requires the solution of non-positive definite systems in connection with solving the eigenvalue problem via inverse iteration. Recently the principal investigator and his colleagues have developed a so-called static-storage partial pivoting scheme for solving indefinite sparse matrix equations. The intention is to incorporate this new algorithm into the testbed, and conduct comparisons with those already available in the testbed. The basic software to implement the algorithm has been completed and tested. The next step is to incorporate it into the testbed in an appropriate way.

A final objective of the proposed research is to extend the methods employed in SPARSPAK for use on multiprocessors. Considerable research and development in this direction has already been done, Some of the algorithms and codes developed by the principal investigator and his colleagues will be installed in the CSM testbed, and performance studies will be conducted to allow comparisons with those already available as part of the testbed. Other algorithms and software may have to be developed.

Symmetric Positive Definite Systems

When A is s.p.d., solving $Ax = b$ typically involves four distinct steps:

1. (Ordering) Find a good ordering for A . That is, a permutation matrix P so that PAP^T has a sparse Cholesky factor L .

2. (Symbolic factorization) Determine the structure of L, and set up a data structure for this factor.

3. (Numerical factorization) Place the elements of A into the data structure, and then compute L.

4. (Triangular solution) Using L , solve the triangular systems $Ly = Pb, L^T z = y$, and then set $x = P^T z$.

SPARSPAK contains state-of-the-art algorithms for dealing with steps 1–4.

Symmetric Positive Definite Systems

One of the key advantages of symmetric positive definite matrices is that Gaussian elimination applied to them does not require interchanges (pivoting) to maintain numerical stability. Since PAP^T is also symmetric and positive definite for any permutation matrix P, this means we can choose to reorder A symmetrically *without regard to numerical stability* and *before the actual numerical factorization begins.*

These options, which are normally not available to us when A is a general indefinite matrix, have enormous practical implications. Since the ordering can be determined before the factorization begins, the locations of the fill suffered during the factorization can also be determined. Thus, the data structure used to store L can be constructed prior to the actual numerical factorization, and spaces for fill components can be reserved. The use of a static data structure allows the data structure to *very* efficient. On average, usually there is ***less that one*** item of overhead storage (pointer, subscript, etc.) for each component of the matrix L.

The computation then proceeds with the storage structure remaining *static* (unaltered). Thus, the three problems of i) finding a suitable ordering, ii) setting up the appropriate storage scheme, and iii) the actual numerical computation, can be isolated as separate objects of study, as well as separate computer software modules.

SPARSPAK - Design Considerations

- Computer programs for solving sparse systems of linear equations typically involve fairly complicated data structures and storage management.
- The unconventional data structures usually mean that subroutines have long argument lists, involving parameters which are of no interest to the user.
- In most cases the user of such programs simply wants to solve the problem, and should not have to understand how the storage management is done, or how the matrix components are actually stored.
- The user should be insulated from these complications, but should still be able to use the package in a variety of ways.
- Ideally, the only information the package should require is that which the user must know anyway.

SPARSPAK - Design Considerations

Computer subroutines for solving dense matrix problems involve conventional numerical data types such as one- or two-dimensional arrays of floating point numbers, which are already available in the programming languages normally used for scientific computation. The storage requirement is known as soon as the dimension of the problem is prescribed, and the number of parameters to such procedures is usually quite modest. In addition, the number of subroutines involved in solving a problem is only one or two. Thus, the "intellectual overhead" in learning how to use the subroutines is quite small.

Unfortunately, very little of this holds for subroutines which implement algorithms for solving sparse systems. The algorithms are relatively complicated, and their implementations typically involve a substantial number of subroutines. The data structures are sufficiently unconventional that they are not provided as standard data types, so subroutines usually have long parameter lists, most of which have no meaning to the user unless he or she cares to know how the data are stored. Finally, the amount of storage is usually unpredictable until at least part of the overall computation has been completed, which complicates the management of computer storage.

SPARSPAK - Features

- Contains implementations of state-of-the-art algorithms.
- Has a friendly *user interface*, which is a layer of software between the user and the numerous subroutines which implement the four phases of the solution procedure.
- The interface provides storage management services, sequencing control, checkpoint and restart facilities, and insulation from long subroutine argument lists.

SPARSPAK - Features

SPARSPAK was designed to address the complications outlined on the previous slide. The package contains implementations of state-of-the-art algorithms, but its novel and unique feature is its *user interface*, which is a layer of software between the user, who has a sparse problem to solve, and the numerous subroutines which implement the four phases of the solution procedure mentioned above. This layer of software provides storage management services, insulates the user from the complicated data structures used by the subroutines, and provides a convenient means of communication between the user and the subroutines. The user is required to know very little more than what he or she must know anyway about the problem to be solved. The interface also provides sequencing control, so that subroutines are called in the correct order, and convenient checkpoint and restart facilities. The latter capability, along with the modular design of the package, allows the user to avoid re-doing parts of the computation if it is aborted after part of it has been successfully completed.

SPARSPAK - Structure of the Package

The user's program and the package interact as follows:

1. (Structure Input) The user supplies the nonzero structure of A.
2. (Order) The package reorders the original problem, (finds a permutation P), and allocates storage for the triangular factorization of PAP^T.
3. (Numerical Input) The user supplies the numerical values for the matrix A to the package.
4. (Factor) The package computes the triangular factors of PAP^T.
5. (Numerical Input) The user supplies numerical values for b. (This step may come before Step 4, and may be intermixed with Step 3.)
6. (Solve) The package computes the solution x, using L, P and b.

SPARSPAK - Structure of the Package

In step one, the user communicates the structure of the matrix to the package by calling one or more simple interface routines. A typical program statement would be $CALL\ INIJ(I,J)$, which would tell the package that there is nonzero element in position (i,j). There are also routines that can be used to communicate the structure corresponding to an entire element stiffness matrix in one call. After the structure is input, a single statement such as $CALL\ ORDER$ invokes an ordering routine (there are several choices), and sets up the data structure.

The user then provides the numerical values to the package by executing a call to one or several routines. These routines allow assembly of the element matrices to be done in a perfectly natural way. The package "knows" when to initialize the data structure, and after that, all input to the package is additive. A typical program statement for numerical input would be $CALL\ INAIJ(I,J,AIJ)$.

After the numerical input is complete, a single call to a subroutine named $SOLVE$ initiates the factorization. If the right hand side has not been input, only the factorization will be performed. If it has, then both factorization and solve will be done. The right hand side is supplied to the package using routines similar to those for inputting the matrix components.

SPARSPAK - Installation in the Testbed

Strategies, issues:

- Leave SPARSPAK more-or-less intact, and implement a single processor that performs the functions of TOPO, K, INV, and SSOL. Inflexible with respect to nonlinear analysis and the eigenvalue problem.
- Decompose SPARSPAK into a number of processors, which serve as alternatives to JSEQ or RSEQ, K, INV and SSOL. More flexible.
- In order to ensure that existing processors can be used in concert with new SPARSPAK-derived ones, it is important that the existing data structures be preserved.

SPARSPAK - Installation in the Testbed

One possibility is to leave SPARSPAK more or less intact. This would imply creating a single processor which would first obtain topological information about the structure from the data base, along with constraint information indicating which variables are constrained. This would then allow SPARSPAK to perform the reordering, set up the data structure for the overall stiffness matrix, and allocate storage for it. The processor would then again access the data base to obtain the element stiffness matrices and applied loads. Using constraint information, the overall assembly could then be performed using the basic facilities that are in SPARSPAK. Once the assembly is complete, factorization and solution can be done.

Alternatively, SPARSPAK could be decomposed into several processors which would serve as alternatives to JSEQ (RSEQ), K, INV, and SSOL.

These alternatives and others are currently being explored with technical personnel at LARC. The first alternative is quite inflexible, and would not lend itself well to many current applications of the testbed. The second provides more flexibility, but if the new processors derived from SPARSPAK are to be used in concert with existing ones, the data/file structures that provide the information exchange among the processors must be maintained.

RSEQ already provides a reordering capability, but it reorders the problem without knowledge of constraints. SPARSPAK will be most efficient if it is allowed to reorder and assemble the matrix *after* the constraints have been applied. One objective is to determine whether exploiting the effect of the constraints is worth the effort. The advantage of the current approach used by RSEQ is that many constraint sets can be considered with only one pass through RSEQ.

Detecting Parallelism in Sparse Matrix Computation

- Objectives:
  - preserve sparsity
  - low arithmetic operation count
  - high parallelism
  - low communication
- These objectives turn out to be complementary.
- Good (serial) orderings are also good for parallel computation, or can be rearranged so that they are good.
- *Elimination trees* are useful in this analysis.

Detecting Parallelism in Sparse Matrix Computation

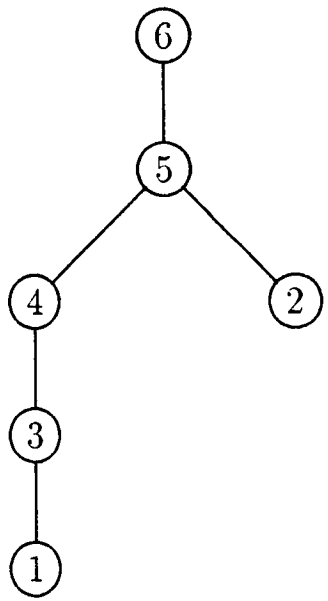
Ideally, we would like to choose an ordering for the matrix A which achieves a number of objectives. First, just as in the use of serial machines, we would like to preserve sparsity and obtain a low arithmetic operation count. In addition, the ordering should allow a high degree of parallelism, and allow the distribution of the computation across the processors in a way that allows the parallelism to be exploited without requiring an inordinate amount of communication.

Fortunately, these objectives turn out to be mutually complementary. In order to gain insight into this problem, it is useful to introduce the notion of elimination trees for sparse Cholesky factors.

Detecting Parallelism - elimination trees

$$L = \begin{pmatrix} \times & & & & & \\ & \times & & & & \\ \times & & \times & & & \\ & & \times & \times & & \\ \times & \times & \times & \times & \times & \\ & \times & & \times & \times & \times \end{pmatrix}$$

Structure of a Cholesky factor



The elimination tree

Detecting Parallelism - elimination trees

Consider the structure of the Cholesky factor L. For each column $j \leq n$, if column j has off-diagonal nonzeros, define $\gamma[j]$ by

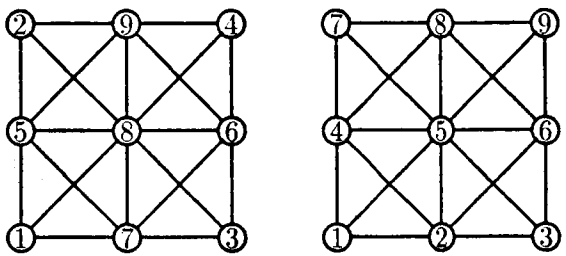
$$\gamma[j] = \min\{i \mid l_{ij} \neq 0, i > j\};$$

that is, $\gamma[j]$ is the row subscript of the first off-diagonal nonzero in column j of L. If column j has no off-diagonal nonzero, we set $\gamma[j] = j$. (Hence $\gamma[n] = n$.)

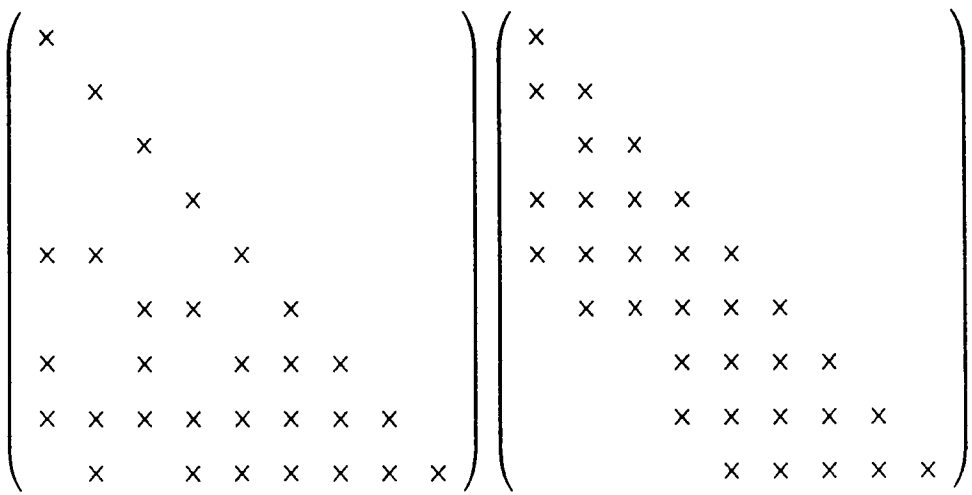
We now define an *elimination tree* corresponding to the structure of L. The tree has n nodes, labelled from 1 to n. For each j, if $\gamma[j] > j$, then node $\gamma[j]$ is the *parent* of node j in the elimination tree, and node j is one of possibly several *child* nodes of node $\gamma[j]$. We assume that the matrix A is *irreducible* , so that n is the only node with $\gamma[j] = j$ and it is the *root* of the tree. Thus, for $1 \leq j < n$, $\gamma[j] > j$.

The importance of the elimination tree is that it gives precise information about the column dependencies. In particular, the computation of column i cannot be completed until the calculation of all columns corresponding to descendent nodes of node i have been completed. On the other hand, columns corresponding to nodes at the same level of the tree are *independent*, and can be computed in parallel.

Elimination trees - an example



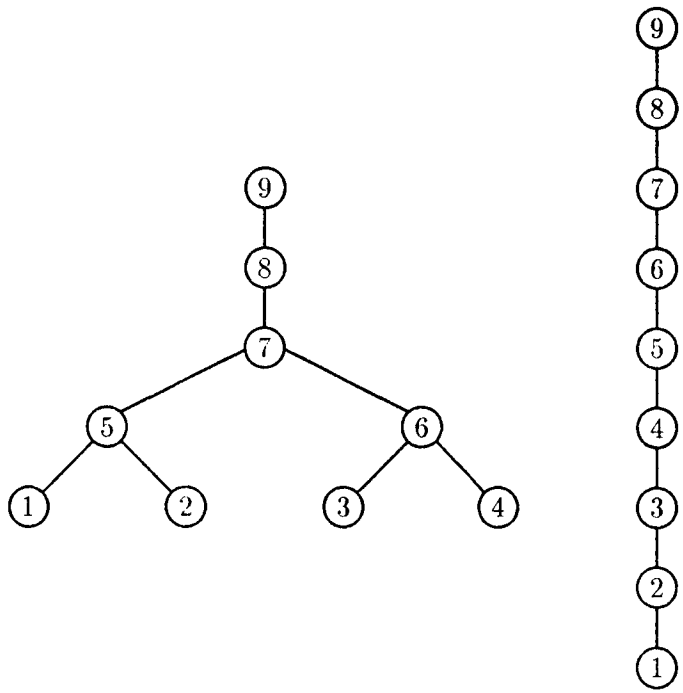
Two orderings and their corresponding factors.



Elimination trees - an example

In order to see the role that elimination trees might play in identifying parallelism, consider two different orderings of a 3 by 3 grid problem. The 9 vertices of the grid are numbered in some manner, and the associated matrix A has the property that $a_{ij} \neq 0$ if and only if vertex i and vertex j are associated with the same small square in the grid. Two different orderings of the grid along with their associated Cholesky factors are given in the chart.

Elimination trees - an example



The elimination trees associated with the matrices.

Elimination trees - an example

The elimination tree on the left is typical of those generated by orderings that are good in the sense of yielding low fill and low operation counts. Its tree structure is short and wide, and such trees and their associated orderings lend themselves well to parallel computation. For example, it should be clear that columns $1, 2, 3$ and 4 can be computed in parallel. Moreover, when they have been computed, columns 5 and 6 can be computed in parallel.

On the other hand, the band-oriented ordering shown above is undesirable because it imposes the same serial execution that is imposed in the dense case. Moreover, the operation counts and fill-in are inferior to that of the first ordering.

In the elimination tree, if node i and node j belong to the same level of the tree, it is clear that the computation of columns i and j can be performed independently so long as the tasks associated with their descendant nodes have all been completed. In order to gain high processor utilization, it is therefore desirable to assign, if possible, nodes on the same level of the tree to different processors.

Elimination trees ... facts

- *All* labellings of the elimination tree that number child nodes before their parents are *equivalent.* They produce the same fill and the same arithmetic operation count.
- Elimination trees can be computed *very* rapidly in serial mode.
- Good orderings may not yield balanced trees.
- Elimination trees can be rearranged (restructured) to enhance the available parallelism $\cdots$ important recent work by Joseph Liu,who has shown how to restructure trees to change their height but preserve the fill and operation counts.

Elimination trees ... facts

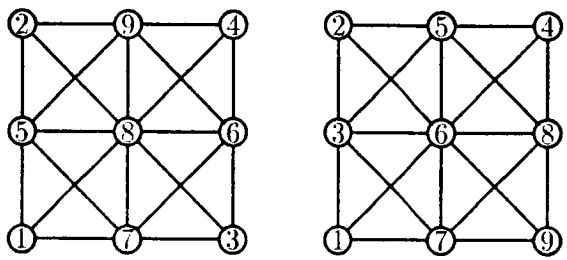
It is generally known (although there seems to be no published proof) that given an ordering of a sparse matrix, and therefore an elimination tree, any symmetric *reordering* of the matrix based on a relabelling of the tree that numbers each vertex ahead of its parent is *equivalent* to the original ordering in terms of fill and computation.

In some contexts it is necessary to be able to obtain the elimination tree prior to determining the structure of L. Fortunately, there is a very efficient algorithm that can be used to obtain the tree directly from the structure of the matrix A due to Liu. Its complexity can be shown to be $O(|A| \log_2 n)$.

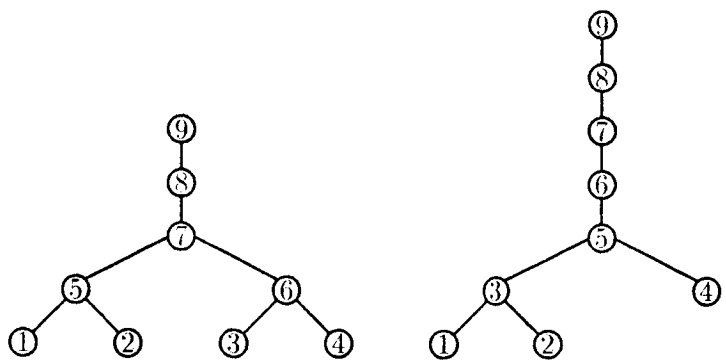
J.W-H. Liu, "Equivalent sparse matrix reordering by elimination tree rotations", Report CS-86-09, Dept of Computer Science, York University.

J.W-H. Liu, "Reordering sparse matrices for parallel elimination", Report CS-87-02, Dept of Computer Science, York University.

Elimination trees - restructuring



Two *equivalent* orderings and their corresponding trees.



Elimination trees ... restructuring

Recently Liu has shown how to produce equivalent reorderings of matrices which *change* the elimination tree. That is, the amount of fill and the amount of computation required for the factorization do not change, but the structure of the tree does change. He has also developed a fast algorithm which will reorder a sparse matrix problem in order to either increase or decrease the height of the elimination tree, while preserving the level of fill and computation.

The chart shows two orderings of the 3×3 grid problem which produce exactly the same fill, but which have different elimination trees. The ordering on the left obviously lends itself to parallel computation somewhat better than the one on the right. In other contexts, such as when auxiliary storage or virtual memory is used, "tall, skinny" trees are desirable because this tends to reduce the amount of main storage that is required. Liu's restructuring algorithm is useful in this situation as well.

J. W-H. Liu, "A note on sparse factorization in a paging environment", Report CS-86-13, Dept. of Computer Science, York University.

An overall strategy

1. Find a good sparsity-preserving ordering for A.
2. Find the elimination tree corresponding to the reordered A.
3. Reorder the problem so as to reduce the height of its elimination tree using Liu's height-reducing algorithm.
4. Assign the column tasks to the processors in a "bottom up" manner with respect to the tree, but as much as possible, assign subtrees of the tree to subsets of the processors.

An overall strategy

In view of the previous observations, it would seem desirable to first find the best ordering in terms of fill and computation, and then within the class of reorderings that preserve that level of fill and computation, choose one which produces a "short" and "wide" tree.

Solving Indefinite Sparse Systems

- Problem to be solved: $Ax = b$.
- If A has no special properties, some form of pivoting is required.
- Important fact: A normally suffers *fill* during the factorization.
- Important fact: permuting the rows and columns of A can have an enormous effect on the amount of fill that occurs.
- Standard solution: Compute a factorization of $P_r A$ or $P_r A P_c$, where P_r and P_c are $n \times n$ permutation matrices. Permutations are determined *during the factorization* by a combination of numerical stability and sparsity requirements.
- Some form of *threshhold pivoting* is usually employed.

Solving Indefinite Sparse Systems

When Gaussian elimination is applied to a sparse matrix A, it normally suffers *fill.* That is, its factors usually have nonzeros in positions which are zero in the corresponding positions in A.

If A has no special properties, it is well-known that applying Gaussian elimination to A may fail. A zero element might turn up in the pivot position. In this case, some form of row and/or column pivoting must be performed in order to ensure numerical stability..

Permuting the rows and columns of A can have an enormous effect on the amount of fill that occurs during the factorization. Note that here and at all times we make the usual *no-cancellation assumption.* That is, whenever two nonzero quantities are added or subtracted, the result is nonzero. This means that in the analysis we ignore any zeros which might be created through exact cancellation. Such cancellation rarely occurs, and any such prediction would be difficult in general, particularly in floating point arithmetic which is subject to rounding errors.

Given A, one normally obtains a factorization of $P_r A$ or $P_r A P_c$, where P_r and P_c are $n \times n$ permutation matrices. When A is sparse, these permutations are determined *during the factorization* by a combination of (usually competing) numerical stability and sparsity requirements. Dynamic data structures obviously are required. Different matrices, even though they may have the same nonzero pattern, will normally yield different P_r and P_c, and therefore have factors with different sparsity patterns.

The use of dynamic data structures tends to lead to very complicated code, and substantial computational and storage overhead.

Solving Indefinite Systems - Partial Pivoting

- Gaussian elimination with partial pivoting yields:

$$A = P_1 M_1 P_2 M_2 \cdots P_{n-1} M_{n-1} U,$$

where P_k is an elementary permutation matrix corresponding to the row interchange performed at step k, M_k is a unit lower triangular matrix whose k-th column contains the multipliers used at the k^{th} step, and U is an upper triangular matrix.

- Structures of $M = \sum_{k=1}^{n-1} M_k$ and U depend on the interchanges, which in turn depends on the numerical values of A.

Solving Indefinite Systems - Partial Pivoting

A standard approach for solving $Ax = b$ involves reducing A to upper triangular form using elementary row eliminations (i.e., Gaussian elimination). In order to maintain numerical stability, one may have to interchange rows at each step of the elimination process. Thus, we may express the result of the process as follows:

$$A = P_1 M_1 P_2 M_2 \cdots P_{n-1} M_{n-1} U,$$

where P_k is an $n \times n$ elementary permutation matrix corresponding to the row interchange performed at step k, M_k is an $n \times n$ unit lower triangular matrix whose k-th column contains the multipliers used at the k^{th} step, and U is an $n \times n$ upper triangular matrix. As noted earlier, when A is sparse, fill normally occurs during the triangular decomposition, so there are usually collectively more nonzeros in $M = \sum_{k=1}^{n-1} M_k$ and U than in A. Moreover, the structures of $M = \sum_{k=1}^{n-1} M_k$ and U depend on the interchanges, which in turn depend on the numerical values of A.

Solving Indefinite Systems - an alternative approach

- Create from the *structure* of A a data structure which can accommodate all the nonzeros in M and U, *irrespective of the actual pivot sequence chosen.* Denote matrices with these structures by $\bar{L}$ and $\bar{U}$.
- Carry out the factorization of A using this static data structure.
- Experience (and some theoretical results) show that the amount by which the data structure is "too big" is not excessive.
- A static data structure is efficient in terms of both space and computation.
- This approach has been implemented and extensively tested.
- One can define an elimination tree for the matrix $\bar{U}$ that can be used to guide the exploitation of parallelism.

Solving Indefinite Systems - an alternative approach

The basic strategy is to create from the *structure* of A a data structure which can accommodate all the nonzeros in M and U, *irrespective of the actual pivot sequence chosen.* Let $\bar{L}$ and $\bar{U}$ be matrices whose structures contain respectively the structures of M and U, irrespective of the pivot sequence $P_1, P_2, \cdots, P_{n-1}$. Although the data structure for these matrices is more generous than it needs to be for any *specific* sequence, it can be set up in advance of the numerical computation, which can then be done efficiently using a static data structure. The advantages noted earlier for positive definite matrices are then be available.

This approach has been implemented and tested, and shown to be very competitive with alternative approaches for solving indefinite finite element problems. Work is currently under way to implement the scheme on a shared memory multiprocessor. One can define an elimination tree for the matrix $\bar{U}$ that can be used to guide the exploitation of parallelism in much the same manner as discussed earlier in the context of solving symmetric positive definite systems on parallel architectures.

Selected Publications

1. Alan George, Michael T. Heath and Joseph W-H. Liu, "Parallel Cholesky Factorization on a Shared-Memory Multiprocessor", Linear Algebra and its Applics., 77 (1986), pp. 165-188.

2. Alan George and Esmond Ng, "Symbolic Factorization for Sparse Gaussian Elimination with Partial Pivoting", SIAM J. Sci. and Stat. Computing, 8 (1987), pp. 877-898.

3. Alan George, Michael T. Heath, Joseph W-H. Liu and Esmond Ng, "Symbolic Cholesky Factorization on a Local-Memory Multiprocessor", Parallel Computing, 5 (1987), pp. 85-95.

4. Eleanor Chu and Alan George, "Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor", Parallel Computing, 5 (1987), pp. 65-74.

5. Alan George, Michael T. Heath, Joseph Liu and Esmond Ng, "Sparse Cholesky Factorization on a Local-Memory Multiprocessor", SIAM J. Sci. and Stat. Computing, (to appear).

6. Alan George and Joseph Liu, "The Evolution of the Minimum Degree Ordering Algorithm", SIAM Review, (to appear).

7. Alan George and Esmond Ng, "On the Complexity of Sparse QR and LU Factorization of Finite Element Matrices", SIAM J. Sci. and Stat. Computing, (to appear).

8. Alan George, Joseph Liu and Esmond Ng, "Communication Results for Parallel Sparse Cholesky Factorization on a Hypercube", Parallel Computing, (submitted).